EECS 280

Error Handling and Exceptions

Handling Errors

- Sometimes a function detects an error but doesn't know what to do about it.
  - Think about going to your boss in an exceptional circumstance.
- We need to separate **error detection** from **error handling**.
  - Usually this means detecting the error in a function, but then letting the caller of that function handle the error.

Example: Error Detection

```
// Opens the file with the given filename and
// returns the contents as a string.
string readFileToString(const string &filename) {

 // Attempt to open the file
 ifstream fin(filename);
 if (!fin.is_open()) {
   // ERROR! Couldn't open file!
   // What should I do!?!
 }
 ...
}
```

Just ignore that file and keep going?

Print a message to cout?

Show a pop-up error message to the user?

Error Handling

- Strategies for communicating an error to the outside world (e.g. a function's caller):
  - Global Error Codes
  - Object Error States
  - Return Error Codes
  - Throw/Catch Exceptions this one is better since
- Again, the idea is that the function that detected an error doesn't have the context to know what should be done.
  - But the caller may know what to do!

Global Error Codes

- Strategy:
  1. Store an error code in a global, then return.
  2. Caller must check the global variable for errors.

After return, the function can't do the rest of job.

- Generally, global anything is poor style.
  - (A fairly reliable rule, at least.)

- In more complex programs, this approach becomes fragile.
  - You have to make sure to check the error code before any other error occurs, otherwise it gets overwritten!

- when we meet an error in ADT (class)
- use public interface to check if the class instance is in the wrong state.

Object Error States

- If a member function fails, it can put the object it was called on into an error state.
- You have to check whether the object is still in a good state after each operation that can fail.

```
...
// Attempt to open the file
ifstream fin(filename);
if (!fin.is_open()) {
 ...
}
...
```

Return Error Codes

- Strategy:

1. Return an error code. "-1"
2. Caller must check the return value for errors.

- Better than the global strategy, because error handling is local and interference is not possible.

- However, now our "error code" must somehow fit into the return value...

Return Error Codes

```
// Returns n! for non-negative inputs
// and returns -1 to indicate an
// erroneous input was detected.
int factorial(int n) {

 // Check for error
 if (n < 0) {
  return -1;
 }
 ...
}
```

这里是在public interface 中输出"-1".
相比于之前我们定义的set, 如果没找到, 我们返回"-1", 但是这一系列返回和实现的操作都是在 implementation 内部进行的. 使用 interface 的人不会对 return "-1" 产生困惑.

Because return negative number is meaningless to factorial. So we can return "-1" to show the error.

OK
A part of the possible return values was "free" to be used.

Return Error Codes

```
// Parses an int from a string.
// Returns the int. Returns ??? to
// indicate an error.
int parseInt(const string &str) {
 // Check for error
 if (/*Bad characters in string*/) {
   return ???;
 }
 ...
}
```

All the possible return values are already filled up with legitimate returns. So there's nothing to use for an error code.

PROBLEM

All the ints we could use for an error code are also legitimate returns.

Return Error Codes

```
// Makes a Duck. If there was a
// problem, returns the WAT duck
// instead.
Duck makeDuck(/*Duck Parameters*/) {

 // Check for error
 if (/*ERROR*/) {
   return Duck("WAT");
 }
 ...
}
```

It's weird to return this to show the "error".

PROBLEM
Sometimes it just feels weird.

Return Error Codes

```
// Parses an int from a string. Returns a
// pair<int,int>. The second member of the pair is
// 0 if there's no error. Otherwise it's an error
// code.
std::pair<int, int> parseInt(const string &str) {
 // Check for error
 if (/*Bad characters in string*/) {
   return {0, 1};
 }
 ...
 return {num, 0};
}
```

对于那些可能为error出现的 function.
return 两个变量
(result) (error code)
error
any junk value, because the user will not use the result. if their is error in function.
success case
result
OK (no error)

OK

Second member in the pair holds the error code.

the "Go" has mechanism to force you check the result code before you can use it.

Some modern languages use this approach, e.g. Go

Other Error Code Issues

- The caller might forget to check for them.

```
// Returns n! for non-negative inputs
// and returns -1 to indicate an
// erroneous input was detected.
int factorial(int n);

int main(int n) {

 int x = askUser();
 int f = factorial(x);
 // Use error code in a computation.
 // Who knows what will happen??
}
```

we have no control of what they type.

error code ↑ is there is an error, and we forget to check, then we use an error code in the programe.

forget check → undefined behavior

Other Error Code Issues

- Error handling code is interleaved with regular control flow. This is poor style.

```
int main() {
 int x = askUser();
 int f = factorial(x);
 if (f < 0) {
  cout << "ERROR" << endl;
 }
 else if (f < 100) {
  cout << "Small factorial" << endl;
 }
 else {
  cout << "Larger factorial" << endl;
 }
}
```

Branches for normal code execution and for error cases are not always easy to tell apart.

error

但是正确的和错误的都混在一起.

normal exception 的 error code 没有 stand out from the normal operation.

导致整个 ← (code 很难读懂)

Using Exceptions

- The **exception** mechanism introduces an additional **control flow path** for error handling.

```
int main() {
  int x = askUser();
  try {
    int f = factorial(x);
    if (f < 100) {
      cout << "Small" << endl;
    }
    else {
      cout << "Larger" << endl;
    }
  }
  catch (const FactorialError &e) {
    cout << "ERROR" << endl;
  }
}
```

Put code that might throw in a try block.

In separate code, we catch the exception and handle the error.

```
class FactorialError { };

// Returns n! for non-negative
// inputs. Throws an exception // on
negative inputs.
int factorial(int n) {

  // Check for error
  if (n < 0) {
    throw FactorialError();
  }
  ...
}
```

When something goes wrong, we throw an exception.

Using Exceptions

- The **exception** mechanism introduces an additional **control flow path** for error handling.

```
int main() {
 int x = askUser();
 try {
  int f = factorial(x);
  if (f < 100) {
   cout << "Small" << endl;
  }
  else {
   cout << "Larger" << endl;
  }
 }
 catch (const FactorialError &e) {
  cout << "ERROR" << endl;
 }
}
```

Put code that might throw in a try block.

In separate code, we catch the exception and handle the error.

```
class FactorialError { };

// Returns n! for non-negative
// inputs. Throws an exception // on
negative inputs.
int factorial(int n) {

 // Check for error
 if (n < 0) {
  throw FactorialError();
 }
 ...
}
```

When something goes wrong, we throw an exception.

Using Exceptions

- The **exception** mechanism introduces an additional **control flow path** for error handling.

```
int main() {
 int x = askUser();
 try {
  int f = factorial(x);
  if (f < 100) {
   cout << "Small" << endl;
  }
  else {
   cout << "Larger" << endl;
  }
 }
 catch (const FactorialError &e) {
  cout << "ERROR" << endl;
 }
}
```

Put code that might throw in a try block.

In separate code, we catch the exception and handle the error.

```
class FactorialError {};

// Returns n! for non-negative
// inputs. Throws an exception // on
negative inputs.
int factorial(int n) {

 // Check for error
 if (n < 0) {
  throw FactorialError();
 }
 ...
}
```

When something goes wrong, we throw an exception.

Using Exceptions

- The **exception** mechanism introduces an additional **control flow path** for error handling.

- The language is essentially providing us with a structured way to...

1. **Detect Errors**: Create and **throw** an error-like object called an exception, which contains information about what happened.
2. Propagate the exception outward from a function to its caller until it is handled.
3. **Handle Errors**: **Catch** the exception in a special block of code that handles the error.

The throw Statement

```
class FactorialError { };

// Returns n! for non-negative
// inputs. Throws an exception
// on negative inputs.
int factorial(int n) {

 // Check for error
 if (n < 0) {
   throw FactorialError();
 }
 ...
}
```

- When a **throw statement** is encountered, regular control flow ceases.
- The program proceeds **outward through each scope** until an appropriate catch is found.
- You can throw any kind of object, but generally we use a class type created to represent a particular kind of error.
  - e.g. FactorialError
- Only one object can ever be thrown at a given time. (No juggling allowed.)

The try-catch block

```
int main() {
 int x = askUser();
 try {
   int f = factorial(x);
   if (f < 100) {
     cout << "Small" << endl;
   }
   else {
     cout << "Larger" << endl;
   }
 }
 catch (const FactorialError &e) {
   cout << "ERROR" << endl;
 }
}
```

- A **try block** is always matched up with one or more **catch blocks**.
- If an exception is thrown inside a try block, the corresponding catch blocks are examined.
- If a catch block matches the type of the exception, the code in that block executes.
- If there is no matching catch, the exception continues outward.
- Uncaught exception == crash.

Exercise: Exception Tracing 1


```
class GoodbyeError { };
void goodbye() {
 cout << "goodbye called\n";
 GoodbyeError e; throw e;
 cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
 cout << "hello called\n";
 goodbye();
 throw HelloError();
 cout << "hello returns\n";
}
```

```
int main() {
 try {
  hello();
  cout << "done\n";
 }
 catch (const HelloError &h) {
  cout << "caught hello\n";
 }
 catch (const GoodbyeError &g) {
  cout << "caught goodbye\n";
 }
 cout << "main returns\n";
}
```

Question **What is the output of main()?**

A) **hello called**
goodbye called
hello returns
caught hello
main returns

B) **hello called**
goodbye called
caught goodbye
main returns

C) **hello called**
goodbye called
done
caught goodbye
main returns

Example: Drive Thru

```
class InvalidOrderException { };

class DriveThru {
public:
 // REQUIRES: The item is on the menu.
 // EFFECTS:  Returns the price for the given item. //
 //           If the item doesn't exist, throws an
 //           InvalidOrderException.
 double getPrice(const string &item) const {
   // YOUR CODE HERE
 }

private:
 // A map from item names to corresponding prices
 std::map<string, double> menu;
};
```

Question

Which of these getPrice functions is correct?

```
template <typename Key_type, typename Value_type>
class Map {
public:
 Value_type& operator[](const Key_type& k);
 Iterator find(const Key_type& k) const;
};
```

non const

```
class InvalidOrderException { };

class DriveThru {
public:
 // REQUIRES: Nothing
 // EFFECTS:  Returns the price for the item.
 //           If the item doesn't exist,
 //           throws an InvalidOrderException.
 double getPrice(const string &item) const {
   // YOUR CODE HERE
 }

private:
 // A map from item names to their prices
 std::map<string, double> menu;
};
```

A

```
double getPrice(const string &item) const {
 auto it = menu.find(item);
 if (it) {
  return it->second;
 }
 else { throw InvalidOrderException(); }
}
```

B

```
double getPrice(const string &item) const {
 auto it = menu.find(item);
 if (it != menu.end()) {
  return it->second;
 }
 else { throw InvalidOrderException(); }
}
```

D

```
double getPrice(const string &item) const {
 if (menu.find(item) != menu.end()) {
  return menu.find(item)->second;
 }
 else { throw InvalidOrderException(); }
}
```

C

```
double getPrice(const string &item) const {
 if (menu.find(item) != menu.end()) {
  return menu[item];
 }
 else { throw InvalidOrderException(); }
}
```

So you can't use "[]" there.

Exercise: Drive-Thru Order (part 2)

- Write a main function that takes an order as a sequence of items from cin and reports the total price.
- If an invalid item is ordered, print a message, but keep going. Stop the order when the user types "done".

```
class InvalidOrderException { };

class DriveThru {
public:
 // EFFECTS:  Returns the price for the given item.
 //           If the item doesn't exist, throws an
 //           InvalidOrderException.
 double getPrice(const string &item) const;
};

int main() {
 DriveThru eats280; // assume this is already initialized for you

 // YOUR CODE HERE
}
```

Question

Which of these main functions works as desired?

C - Neither

exceptions are used for normal program behavior.

user点菜单上没有的菜是意料之中的，但如果在菜单里报错，那就是bug.

A

```
int main() {
 DriveThru eats280; // assume this is initialized

 double total = 0; string item;
 try {
   while (cin >> item && item != "done") {
     total += eats280.getPrice(item);
   }
 }
 catch (const InvalidOrderException &e) {
   cout << "Sorry, we don't have: " << item << endl;
 }
 cout << "Your total cost is: " << total << endl;
}
```

When we didn't find the item on menu, we will throw an exception to "try", then catched by "catch", then directly go to the last line and calculate the total

B

```
int main() {
 DriveThru eats280; // assume this is initialized

 double total = 0; string item;
 while (cin >> item && item != "done") {
   try {
     total += eats280.getPrice(item);
   }
   catch (const InvalidOrderException &e) {
     cout << "Sorry, we don't have: " << item << endl;
   }
 }
 cout << "Your total cost is: " << total << endl;
}
```

throw an exception. if this is not on the menu,

Exceptions Example


```
int main() {
 try {
  gradeSubmissions();
  cout << "Grading done!" << endl;
 }
 catch (const csvstream_exception &e) {
  cout << e.what() << endl;
  return 1;
 }
}
```

```
void gradeSubmissions() {
1: vector<string> students = loadRoster();
  for (const string &s : students) {
   try {
2.1: auto sub = loadSubmission(s);
2.2: double result = grade(sub);
2.3: emailStudent(s, result);
   }
   catch (const FileError &e) {
     cout << "Can't grade: " << s << endl;
   }
   catch (const EmailError &e) { ... }
  }
}
```

```
class FileError { };
class EmailError { };
```

```
vector<string> loadRoster() {
 // If the file couldn't be opened,
 // csvstream ctor throws an exception.
 csvstream csvin("280roster.csv");

 // Read in and return the roster
}
```

```
Submission loadSubmission(
              const string &id) {

 // Attempt to open student files

 if (/* can't open files */) {
  throw FileError();
 }

 // Create Submission object from
 // files and return it.
}
```

a loop

several steps for each student.

skip, because the exception

grade next student

escape from loadSubmission function

In all cases, we need consider what is the right level of my program to catch and handle an exception.

Exceptions Example

还是 return error message. 由main()来决定是 return 1;

但是我们将做决定的权力留给了 main()

```
int main() {
  try {
    gradeSubmissions();
    cout << "Grading done!" << endl;
  }
  catch (const csvstream_exception &e) {
    cout << e.what() << endl;
    return 1;
  }
}
```

a function give you access to an internal error message that the csv exception stored

If the Rosten is wrong, the escape

```
void gradeSubmissions() {
  vector<string> students = loadRoster();
  for (const string &s : students) {
    try {
      auto sub = loadSubmission(s);
      double result = grade(sub);
      emailStudent(s, result);
    }
    catch (const FileError &e) {
      cout << "Can't grade: " << s << endl;
    }
    catch (const EmailError &e) { ... }
  }
}
```

grade Submissions() really can't do it's job.

So that exception would come out of the grade Sumission

```
class FileError { };
class EmailError { };
```

```
vector<string> loadRoster() {
  // If the file couldn't be opened,
  // csvstream ctor throws an exception.
  csvstream csvin("280roster.csv");
  // Read in and return the roster
}
```

if it can't open the file, it will throw an exception

an istance of a class called: csvstream_exceptio

```
Submission loadSubmission(
              const string &id) {

  // Attempt to open student files

  if (/* can't open files */) {
    throw FileError();
  }

  // Create Submission object from
  // files and return it.
}
```

Custom Exception Types

- DO: Use custom exception types.
  - The type itself indicates the kind of error.
  - The thrown object may also carry along extra information.

```
class EmailError : public std::exception {
public:
 EmailError(const string &msg_in) : msg(msg_in) { }
 const char * what() const override { return msg.c_str(); }
private:
 string msg;
};
```

Best practice is to derive from std::exception.

Override what() member function to retrieve message.

```
throw EmailError("Error sending email to: " + address);
```

- Always use catch-by-reference (to const).

```
try { ... }
catch (const EmailError &e) {
 cout << e.what() << endl;
}
```

- DO NOT: Throw "regular" types (e.g. int, string, vector).

Exceptions and Polymorphism

- It is common to define a hierarchy of exception types, which can be caught polymorphically.

```
class EmailError : public std::exception { ... };
class InvalidAddressError : public EmailError { ... };
class SendFailedError : public EmailError { ... };
```

```
void gradeSubmissions() {
 vector<string> students = loadRoster();
 for (const string &s : students) {
  try {
   auto sub = loadSubmission(s);
   double result = grade(sub);
   emailStudent(s, result);
  }
  catch (const FileError &e) {
   cout << "Can't grade: " << s << endl;
  }
  catch (const EmailError &e) { ... }
 }
}
```

This catches any kind of EmailError. Note the catch by reference is necessary for polymorphism.

Multiple Catch Blocks

Recall:

- Catch blocks are tried in order.
- The **first matching block** is used.
- At most **one** catch block will ever be used.
- If none match, the exception continues outward.

```
try {

 // Some code that may throw many different kinds of exceptions

}
catch (const InvalidAddressError &e) {
 cout << e.getMessage() << endl;
 // Also, remove the recipient from our address book
}
catch (const EmailError &e) {
 cout << "Error sending mail: <<  e.getMessage() << endl;
}
catch (const SomeOtherError &e) {
 // Do something to hand
}
catch (...) {
 cout
```

First, attempt to match a specific kind of email error.

Match any remaining email errors.

Because of the rule that we only execute one of the catch blocks, the first one that matches,

Writing ... will match anything.

Because you want to respond to specific kinds of errors

This last catch with ... is a bad idea. Why?

To catch or not to catch?

like you can actually take some action that puts your program back on track.

- Only catch an exception if you can responsibly handle it.

```
void gradeSubmissions() {
 vector<string> students = loadRoster();
 for (const string &s : students) {
  try { /* Open files, grade submission, email student */ }
  catch (const FileError &e) {
   cout << "Can't grade: " << s << endl;
  }
 }
}
```

In this context, just logging an error message and moving on is reasonable.

- Don't catch an exception if you don't know how to handle it and still "do your job" successfully.

```
vector<string> loadRoster() {
 try {
  csvstream csvin("280roster.csv"); // ctor may throw
  // Use the stream to load the roster...
 }
 catch (const csvstream_exception &e) {
  cout << e.what() << endl;
 }
}
```

If the csvstream fails, we can't do our job (load/return a vector).

Instead, we should NOT catch the error here and allow the exception to propagate out of the function.

Exercise: Exception Tracing 2

- What is the output of this code?

```
class GoodbyeError { };
void goodbye() {
 cout << "goodbye called\n";
 GoodbyeError e; throw e;
 cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
 cout << "hello called\n";
 try { goodbye(); }
 catch (const GoodbyeError &ge) {
   throw HelloError();
 }
 cout << "hello returns\n";
}
```

```
int main() {
 try {
   hello();
   cout << "done\n";
 }
 catch (const HelloError &he) {
   cout << "caught hello\n";
 }
 catch (const GoodbyeError &ge) {
   cout << "caught goodbye\n";
 }
 cout << "main returns\n";
}
```

Solution: Exception Tracing 2

- What is the output of this code?

```
class GoodbyeError { };
void goodbye() {
 cout << "goodbye called\n";
 GoodbyeError e; throw e;
 cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
 cout << "hello called\n";
 try { goodbye(); }
 catch (const GoodbyeError &ge) {
   throw HelloError();
 }
 cout << "hello returns\n";
}
```

```
int main() {
 try {
   hello();
   cout << "done\n";
 }
 catch (const HelloError &he) {
   cout << "caught hello\n";
 }
 catch (const GoodbyeError &ge) {
   cout << "caught goodbye\n";
 }
 cout << "main returns\n";
}
```

```
hello called
goodbye called
caught hello
main returns
```

Exercise: Exception Tracing 3

- What is the output of this code?

```
class Error {
 string msg;
public:
 Error(const string &s) : msg(s) { }
 const string &get_msg() { return msg; }
};
```

```
void goodbye() {
 cout << "goodbye called\n";
 throw Error("bye");
 cout << "goodbye returns\n";
}
```

```
void hello() {
 cout << "hello called\n";
 try { goodbye(); }
 catch (const Error &e) { throw Error("hey"); }
 cout << "hello returns\n";
}
```

```
int main() {
 try {
  hello();
  cout << "done\n";
 }
 catch (const Error &e) {
  cout << e.get_msg();
  cout << endl;
 }
 catch (...) {
  cout << "unknown error\n";
 }
 cout << "main returns\n";
}
```

Solution: Exception Tracing 3

- What is the output of this code?

```
class Error {
 string msg;
public:
 Error(const string &s) : msg(s) { }
 const string &get_msg() { return msg; }
};
```

```
void goodbye() {
 cout << "goodbye called\n";
 throw Error("bye");
 cout << "goodbye returns\n";
}
```

```
void hello() {
 cout << "hello called\n";
 try { goodbye(); }
 catch (const Error &e) { throw Error("hey"); }
 cout << "hello returns\n";
}
```

```
int main() {
 try {
  hello();
  cout << "done\n";
 }
 catch (const Error &e) {
  cout << e.get_msg();
  cout << endl;
 }
 catch (...) {
  cout << "unknown error\n";
 }
 cout << "main returns\n";
}
```

hello called
goodbye called
hey
main returns

Exercise: Exception Tracing 4

- What is the output of this code?

```
class Error {
 string msg;
public:
 Error(const string &s) : msg(s) { }
 const string &get_msg() { return msg; }
};
```

```
void goodbye() {
 cout << "goodbye called\n";
 throw GoodbyeError();
 cout << "goodbye returns\n";
}
```

```
void hello() {
 cout << "hello called\n";
 try { goodbye(); }
 catch (const Error &e) { throw Error("hey"); }
 cout << "hello returns\n";
}
```

```
int main() {
 try {
  hello();
  cout << "done\n";
 }
 catch (const Error &e) {
  cout << e.get_msg();
  cout << endl;
 }
 catch (...) {
  cout << "unknown error\n";
 }
 cout << "main returns\n";
}
```

Solution: Exception Tracing 4

- What is the output of this code?

```
class Error {
 string msg;
public:
 Error(const string &s) : msg(s) { }
 const string &get_msg() { return msg; }
};
```

```
void goodbye() {
 cout << "goodbye called\n";
 throw GoodbyeError();
 cout << "goodbye returns\n";
}
```

```
void hello() {
 cout << "hello called\n";
 try { goodbye(); }
 catch (const Error &e) { throw Error("hey"); }
 cout << "hello returns\n";
}
```

```
int main() {
 try {
  hello();
  cout << "done\n";
 }
 catch (const Error &e) {
  cout << e.get_msg();
  cout << endl;
 }
 catch (...) {
  cout << "unknown error\n";
 }
 cout << "main returns\n";
}
```

hello called
goodbye called
unknown error
main returns